Computing and Informatics Journal, Vol. 20, 2001, 1–26

STRATEGIES OF STRUCTURAL SYNTHESIS OF PROGRAMS AND ITS EXTENSIONS

Mihhail Matskin
Enn Tyugu

Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway
misha@idi.ntnu.no

Department of Teleinformatics,
Royal Institute of Technology,
Stockholm, Sweden
tyugu@it.kth.se

Abstract.
Proof search for the structural synthesis of programs (SSP) - a deductive program synthesis method which is suited for compositional programming in large and is in practical use in a number of programming environments is explained. SSP is based on a decidable logical calculus where complexity of the proof search is still PSPACE. This requires paying special attention to the efficiency of search. The practice of application of SSP has given its several modifications and extensions. Besides the general case of SSP and its strategies, we present synthesis with independent subtasks, a number of heuristics used for speeding up the search and partial deduction in the framework of SSP.

Keywords. Program construction, proof search startegies, structural synthesis of programs.

1 INTRODUCTION

Structural synthesis of programs has been known for quite a number of years and has been used at least in two commercial systems: XpertPriz and PRIZ [14, 13]. It is the basis for programming in the NUT system [18]. During the years, SSP has gone through modifications, and a number of search strategies have been implemented whose descriptions are difficult to find in print. There are new implementations of SSP done in several places, e.g. [4]. In order to help the new implementors to achieve the best performance of the synthesis we presentt here the startegies of SSP developed and fine-tuned over the years. The present paper describes intuitively the idea of SSP and thereafter describes its variations which differ in complexity of proof search.

We explain here in a comprehensive way how SSP works and how it can be practically used in applications. SSP is based on the idea that we can construct programs taking into account only their structural properties. We use this idea for constructing programs from small as well as large modules whose precise behavior would be difficult to describe in detail. Each preprogrammed module is supplied with an axiom telling under which conditions it can be applied and which values it computes. However, the axiom doesn't describe the relation between the input and output values. Instead of an axiom of the form $\forall x P(x) \rightarrow \exists y R(x, y)$ with precondition P and postcondition R we shall use $\forall x P(x) \rightarrow \exists y R(y)$ (where x is not present in R) which reduces to $(\exists x P(x)) \rightarrow \exists y R(y)$. This axiom tells us that a correct value (satisfying R) can be computed without showing its relation to the input value (which satisfies P). As a consequence, we can represent axioms in a propositional language (considering the closed formulae $\exists x P(x)$ and $\exists y R(y)$ as propositions whose internal structure is inessential). This allows us to use a logic where we are able to handle theories (specifications for synthesis of programs) with large amount (up to tens of thousands) of axioms.

Let us have a computational problem, i.e. a problem of computing desired output from a given input. The problem is described for SSP by a set of formulae (specific axioms) extracted from a specification in a high-level language, and a goal is formulated as a theorem to be proven. Then a proof of the theorem is built and a program for solving the problem is extracted from the proof.

This schema is common for deductive program synthesis (see, for example, [8, 7]). SSP assumes that realizations of axioms are given in some language outside the logic. This provides simplicity and efficiency of the approach and, at the same time, allows us to cover a large number of practical cases.

The rest of the paper is organized as follows. First we define logical basis of SSP. Then we introduce the main SSP strategy and prove its soundness and completeness. Thereafter we describe an object-oriented framework for SSP and discuss some strategies based on the object-oriented approach. Finally, a partial deduction strategy for SSP is presented.

2 STRUCTURAL SYNTHESIS OF PROGRAMS

Here we describe the logic of structural synthesis, its relation to a specification language, and discuss its properties. In brief: this logic is an implicative fragment of the intuitionistic propositional calculus (IPC).

2.1 Logical language

Definition 2.1. (*LL* language) *LL* language (the logical language of SSP) has only the following three kinds of formulae:

1. Propositional variables.
   They are denoted by identifiers beginning with capital letters, for example: A, B, C, D, A^i, A_i ....
2. Unconditional computability statements:
   $$A_1 \& \ldots \& A_k \to B$$
   We also use $\underline{A}$ as an abbreviation for $A_1 \& \ldots \& A_k$
3. Conditional computability statements
   $$(\underline{A}^1 \to B^1) \& \ldots \& (\underline{A}^n \to B^n) \to (\underline{C} \to D)$$
   We also use $(\underline{\underline{A} \to B})$ as an abbreviation for $(\underline{A}^1 \to B^1) \& \ldots \& (\underline{A}^n \to B^n)$.

Computational meaning of the *LL* formulae can be described as follows.

Propositional variables of *LL* correspond to object variables from the source problem description language (specification language). For example, if we have an object variable a in the problem description then there is a propositional variable A in the logical language that expresses computability of a value of the object variable a.

An unconditional computability statement $\underline{A} \to B$ expresses a computability of the value of the object variable b corresponding to B from values of $a_1 \ldots a_k$ corresponding to $\underline{A}$.

A conditional computability statement, e.g. $(A \to B) \to (C \to D)$ expresses computability of the object variable d from c depending on other computations, here depending on the computation of the object variable b from a.

We would like to stress the fact that the *LL* formulae represent only computability of object variables, not relations on their particular values.

We can compare axioms in the form of conditional computability statements, e.g. $(A \to B)\&...\&(C \to D) \to (F \to G)$ with clauses in Prolog [5]. The implication $F \to G$ shows when the axiom can be useful - it corresponds to the head of the clause. The implications $A \to B, ..., C \to D$ represent subgoals (we shall call them subtasks) which have to be solved for satisfying the goal $F \to G$. However, these subgoals are used in a more general way than subgoals in the body of a Prolog clause. In fact, the realization of the axiom can be an arbitrarily complex program that calls synthesized realizations of the subgoals in any suitable order and as many times as needed. Just this provides generality to the structural synthesis of programs: one can preprogram any kind of composition of subtasks.

2.2 Inference rules

In order to guarantee constructiveness of proofs, the SSP uses intuitionistic logic. It is sufficient to use introduction and elimination rules for implication and conjunction only, because these are the only logical connectives of the LL. However, in this case we have to accept conjunctions of propositional variables and of simple implications as formulae of our logic as well. These conjunctions will appear in the process of derivation. The natural deduction rules for handling implications and conjunctions are the following. (We shall use bold capital letters as metasymbols for denoting formulae.)

(& +)

$$\frac{\mathbf{A} \qquad \mathbf{B}}{\mathbf{A\&B}}$$

(&-)

$$\frac{\mathbf{A\&B}}{\mathbf{A}} \qquad \frac{\mathbf{A\&B}}{\mathbf{B}}$$

(→ +)

$$\begin{array}{c} [\mathbf{A}] \\ \vdots \\ \dfrac{\mathbf{B}}{\mathbf{A} \rightarrow \mathbf{B}} \end{array}$$

(→ -)

$$\frac{\mathbf{A} \rightarrow \mathbf{B} \qquad \mathbf{A}}{\mathbf{B}}$$

2.3 Completeness of LL

It is somewhat surprising that LL with its simple implicative formulae is in the sense of derivability equivalent to the whole intuitionistic propositional language. This intuitionistic completeness of LL can be proven (see [10]) by reducing arbitrary intuitionistic propositional formulae to LL-form by introducing new variables.

Using the equivalence replacement theorem $(X \leftrightarrow \mathbf{F}) \rightarrow (\mathbf{G} \leftrightarrow \mathbf{G_F}[X])$, where $\mathbf{G_F}[X]$ is the result of substitution of X for $\mathbf{F}$ in $\mathbf{G}$ one can substitute new variables for subformulas, and reduce the depth of nestedness of a formula as well as eliminate disjunction, equivalence and negation connectives in it, transforming the formula into the form of LL. To preserve the derivability, one has to add new axioms given in the Table 1 when substituting X instead of a formula $A\&B$, $A \vee B$ or $A \rightarrow B$. As we can see, the new axioms are in the form of LL formulas, except one axiom with disjunction.

Table 1.

F	new axioms
$A\&B$	$X \to A, X \to B, A\&B \to X$
$A \vee B$	$X \to A \vee B, A \to X, B \to X$
$A \to B$	$X\&A \to B, (A \to B) \to X$

The axiom with disjunction $X \to A \vee B$ can be replaced with the set of formulae $X\&(A \to G)\&(B \to G) \to G$ for all propositional variables G of the theory. This is justified by the second order equivalence $(p \vee q) \leftrightarrow \forall x((p \to x)\&(q \to x) \to x)$.

The negation $\neg A$ can be represented by implication $A \to \bot$ where $\bot$ is the falsity constant. The falsity constant, in its turn, can be replaced by the new variable L and formula $L \leftrightarrow K\&Z$ where Z is a new variable and K is the conjunction of all propositional variables of the theory. This is justified by the second order equivalence $\bot \leftrightarrow \forall xx$.

An experiment has been done with a programming system based on SSP (see [17, 11]). In this experiment, all intuitionistic propositional theorems (about 100 formulae) contained in Kleene's "Introduction to Metamathematics" [2] were encoded in LL using the transformations described above and proven automatically.

2.4 Program extraction

Let us look at the computational meaning of formulas of LL. We use the Kleene realizability of intuitionistic formulas here, and associate a term (a program) with each derivable formula. In order to be able to extract programs from proofs, the language LL is extended to the language $LL1$ as follows.

Definition 2.2. (*LL*1 language) $LL1$ language is an extension of the LL language where it is possible to show explicitly a realization of a formula. In addition to the LL formulae, it includes the following kinds of formulae:

1. Propositional variables with realization: $A(a)$ for denoting the fact that a is a realization of the propositional variable A.
2. Unconditional computability statements with realization:

$$\underline{A} \underset{f}{\to} B$$

   where f is a term representing the function which computes the realization b of B from the realization a of A.
3. Conditional computability statements with realization:

$$\underline{(\underline{A} \underset{g}{\to} B)} \to (\underline{C} \underset{F}{\to} D)$$

where g is a variable representing a function which must be synthesized in order to compute b from $\underline{a}$ and F is a term representing computation of d depending on $\underline{g}$ and $\underline{c}$ ($\underline{g}$ is a tuple of variables for subtasks). It has the form $\lambda \underline{g}\underline{c}.F'$ in λ-notation.

4. Conjunctions of propositional variables and simple implications with or without realizations. (These conjunctions may appear as results of intermediate derivation steps.)

Program extraction occurs step by step in the following way. Realizations of axioms are given. Realization of an assumption X is a variable x. Realization of a derived formula is built from realizations of premises of its inference rule as shown below.

(& +)

$$\frac{\mathbf{A}(a) \qquad \mathbf{B}(b)}{\mathbf{A}\&\mathbf{B}(a,b)}$$

(&-)

$$\frac{\mathbf{A}\&\mathbf{B}(a,b)}{\mathbf{A}(a)} \qquad \frac{\mathbf{A}\&\mathbf{B}(a,b)}{\mathbf{B}(b)}$$

($\rightarrow$ +)

$$\begin{array}{c}[\mathbf{A}(x)]\\ \vdots\\ \dfrac{\mathbf{B}(b)}{\mathbf{A} \rightarrow \mathbf{B}(\lambda x.b)}\end{array}$$

($\rightarrow$ -)

$$\frac{\mathbf{A} \rightarrow \mathbf{B}(f) \qquad \mathbf{A}(a)}{\mathbf{B}(f(a))}$$

Calculus that is based on conventional inference rules of IPC as presented above is practically unsuitable for program synthesis for the following two reasons. First, the extracted program will contain unnecessary steps of composition and decomposition of data structures that correspond to the realizations of conjunctions appearing in a derivation. Second, the derivation steps are small and it is difficult to perform a goal-oriented search using the conventional inference rules. In the next section we introduce new inference rules and describe a calculus immediately suitable for program synthesis.

3 THE MAIN SSP STRATEGY

The main SSP strategy is based on the idea of using the computability information present in axioms for directing the search. It can be explained in two parts. First, instead of the conventional IPC rules we use admissible inference rules each of whose corresponds exactly to an application of a preprogrammed module represented by an axiom, i.e. corresponds to a computational step performed by calling a program that is the realization of the axiom. (For conditional computability statements, it is much more than is done at the implication elimination step.) Second, the selection of axioms is done systematically in such a way that for unconditional computability statements the complexity of search is linear in time and for conditional computability statements it is a goal-directed search on an and-or search tree. Now we present the inference rules for LL.

3.1 Structural synthesis rules

We use a sequent notation for the inference rules of SSP. A sequent $\Gamma \vdash \mathbf{X}$, where Γ is a list of formulae (we use capital Greek letters for denoting a list of formulae) and $\mathbf{X}$ is a formula of LL means that the formula $\mathbf{X}$ is derivable from the formulae appearing in Γ. Logical axioms have a form of $\Gamma, \mathbf{X} \vdash \mathbf{X}$.

Definition 3.1. (SSP inference rules) There are the following inference rules for LL that support the main SSP strategy.

($\rightarrow$ -)

$$\frac{\Sigma \vdash \underline{A} \rightarrow V; \qquad \underline{\Gamma \vdash A}}{\Sigma, \underline{\Gamma} \vdash V}$$

where $\underline{\Gamma \vdash A}$ is a list of sequents $\Gamma^i \vdash A^i$, one for each A in $\underline{A}$

($\rightarrow$ +)

$$\frac{\Gamma, \underline{A} \vdash B}{\Gamma \vdash \underline{A} \rightarrow B}$$

($\rightarrow$ - -)

$$\frac{\Sigma \vdash \underline{(\underline{A} \rightarrow B)} \rightarrow (\underline{C} \rightarrow V); \qquad \underline{\Gamma, \underline{A} \vdash B}; \qquad \underline{\Delta \vdash C}}{\Sigma, \underline{\Gamma}, \underline{\Delta} \vdash V}$$

where $\underline{\Gamma, \underline{A} \vdash B}$ and $\underline{\Delta \vdash C}$ are lists of sequents for B-s in $\underline{B}$ and C-s in $\underline{C}$ respectively.

3.2 Soundness and completeness of the main SSP strategy

Let us remind normal forms of derivations and the normal form theorem first (see also [10])

Definition 3.2. (Normal form) A deduction is in normal form iff it contains only subformulas of its endsequent. The inference rules are applied only for the connectives occurring in the endsequent and they follow the signs of occurrences: introduction rules are applied for positive and elimination rules for negative occurrences.

Definition 3.3. (Long normal form) A deduction is in long normal form iff it is in normal form and the conclusion of any elimination rule having a form $A \to B$ or $A\&B$ is itself a leftmost premise of an elimination rule.

Theorem 3.1. (Normal form theorem) Any deduction (in IPC) can be transformed into the long normal form deduction of the same sequent.

It is easy to check that the SSP rules are admissible rules of the IPC. We show it for the rule $(\to --)$ only. Instead of using the rule $(\to --)$ one can use the following derivation containing only the conventional rules for introduction and elimination of implication:

$$\dfrac{\dfrac{(\underline{A} \to B) \to (\underline{C} \to V) \vdash (\underline{A} \to B) \to (\underline{C} \to V); \quad \dfrac{\Gamma, \underline{A} \vdash B}{\Gamma \vdash \underline{A} \to B}(\to +)}{(\underline{A} \to B) \to (\underline{C} \to V), \Gamma \vdash \underline{C} \to V}(\to -) \qquad \Delta \vdash \underline{C}}{(\underline{A} \to B) \to (\underline{C} \to V), \Gamma, \Delta \vdash V}(\to -)$$

Theorem 3.2. If a sequent in LL has a deduction in the long normal form, then it has a deduction by means of SSP rules.

Proof. Consider a long normal form deduction of a sequent $\mathbf{C_1}, ..., \mathbf{C_n} \vdash A_1\&...\&A_k \to B$ where $\mathbf{C_i}$ are computability statements or variables, and A_i, B are variables. It is sufficient to prove the theorem for $k = 0$, since the sequent can be replaced by $\mathbf{C_1}, ...\mathbf{C_n}, A_1, ..., A_k \vdash B$.

Due to subformula property, the only antecedent members in the deduction are $\mathbf{C_i}$ and variables. Let us consider some uppermost application of an elimination rule. Its left premise is an axiom $\mathbf{C_i} \vdash \mathbf{C_i}$ for some i.

If this is an unconditional computability statement $A_1\&...\&A_n \to B$ then the part of the deduction down is of the form (we use an equivalent representation $A_1 \to (...(A_k \to B)...)$ of the formula $A_1\&...\&A_n \to B$)

$$\dfrac{\mathbf{C_i} \vdash A_1 \to (...(A_k \to B)...) \qquad \Gamma_1 \vdash A_1}{\mathbf{C_i}, \Gamma_1 \vdash A_2 \to (...(A_k \to B)...) \qquad \Gamma_2 \vdash A_2}$$

$$\vdots$$

$$\frac{\mathbf{C_i}, \Gamma_1 ... \Gamma_{k-1} \vdash A_k \to B \qquad \Gamma_k \vdash A_k}{\mathbf{C_i}, \underline{\Gamma} \vdash B}$$

and it can be easily transformed into the application of $(\to -)$.

If $\mathbf{C_i}$ is a conditional computability statement, then the part of the deduction is of the form

$$\frac{\mathbf{C_i} \vdash (\underline{(\underline{A} \to B)} \to (\underline{C} \to V) \qquad \Gamma \vdash \underline{\underline{(\underline{A} \to B)}}}{}$$

$$\frac{\mathbf{C_i}, \Gamma \vdash \underline{C} \to V \qquad \Sigma \vdash \underline{C}}{\mathbf{C_i}, \Gamma, \Sigma \vdash V}$$

where the lower level has the same form as for the unconditional computability statement. Handling a derivation of each subtask $\underline{A} \to B$ and changing it into derivation of B we turn the derivation into the application of $(\to --)$.

Since there are no other applications of introduction rules besides those we have considered in this fragment then we have described the procedure for transforming the long normal derivation into the SSP derivation, that proves our theorem.

Q.E.D.

3.3 Example of a proof

Let us show a proof of the well-known Kripkes formula $((((A \to B) \to A) \to A) \to B) \to B$ which is an intuitionistic analog of the following classically valid formula $((A \to B) \to A) \to A$. This example demonstrates also the idea of encoding of arbitrary intuitionistic propositional formulae in LL.

First we denote

$(A \to B)$ by $X1$,

$(X1 \to A)$ by $X2$,

Next step is applying the transformation (substitution of a new variable X instead of a subformula $(G \to H)$) from the Section 2.3 $(\to)$ to the implications above and to Kripke's formula. The resulting axioms include the following sequents:

for $X1$:

$\vdash (A \to B) \to X1; \vdash X1 \& A \to B$

for $X2$:

$\vdash (X1 \to A) \to X2; \vdash X1\&X2 \to A$

for the formula $(((A \to B) \to A) \to A) \to B$;

$\vdash (X2 \to A) \to B$

These axioms are sufficient for deriving the goal (which is the right side of the outermost implication in the original formula):

$\vdash B$

For brevity, here and further on in examples we use specific (problem oriented) axioms with empty antecedent, i.e. instead of $\mathbf{X} \vdash \mathbf{X}$ we use $\vdash \mathbf{X}$ in the derivations.

A proof in our logic is the following:

$$\frac{\vdash (X2 \to A) \to B; \quad X2, A \vdash A;}{\frac{\vdash (A \to B) \to X1); \quad A \vdash B;}{\frac{\vdash X1\&X2 \to A; \quad \vdash X1; X2 \vdash X2;}{\frac{\vdash (X2 \to A) \to B; \quad X2 \vdash A;}{\vdash B;}(\to --)}(\to -)}(\to --)}(\to --)$$

Q.E.D.

3.4 Program extraction for admissible rules

To facilitate program extraction we use inference rules decorated with realizations showing how realization of a new formula is built. The admissible inference rules decorated with realizations are as follows.

Definition 3.4. (SSP inference rules for $LL1$ language) There are the following inference rules in the LL1:

$(\to$ -)

$$\frac{\Sigma \vdash \underline{A} \underset{f}{\to} V; \quad \underline{\Gamma \vdash A(a)}}{\Sigma, \underline{\Gamma} \vdash V(f(\underline{a}))}$$

$(\to +)$

$$\frac{\Gamma, \underline{A} \vdash B(b)}{\Gamma \vdash \underline{A} \xrightarrow[\lambda \underline{a}.b]{} B}$$

$(\to - -)$

$$\frac{\Sigma \vdash \underline{(\underline{A} \xrightarrow[g]{} B)} \to (\underline{C} \xrightarrow[F]{} V); \quad \underline{\Gamma, \underline{A} \vdash B(b)}; \quad \underline{\Delta \vdash C(c)}}{\Sigma, \underline{\Gamma}, \underline{\Delta} \vdash V(F(\underline{\lambda \underline{a}.b}, \underline{c}))}$$

Construction of the lambda-terms in the rules follows immediately from the soundness proof of the rules (see Section 3.2).

3.5 Selection of axioms and implementation tips

By introducing the admissible rules that correspond to application of program modules we have reduced the search needed for building a proof of solvability of a problem to a search for a proper sequence of application of axioms representing the modules. Here we apply the following tactics:

First look for an applicable unconditional computability statement and apply it.

If there are no more applicable unconditional computability statements, try to apply some conditional computability statement.

3.5.1 Selection of unconditional computability statements

An unconditional computability statement is applicable, iff all its inputs are derived and its output is not yet derived. In order to check this applicability condition efficiently, a data structure with a counter of not yet derived inputs for each computability statement is used. This technique is well known as value propagation in constraint programming. For the program synthesis it was described for the first time in [1]. Using counters together with pointers between the propositional variables and axioms where a variable occurs guarantees that no occurrence will be checked more than once, consequently, the time for search will be linear with respect to the size of a theory.

3.5.2 Pruning

Value propagation is a forward search tactic that produces in general a derivation with redundant steps - any applicable axiom may be included into the derivation, even if it is not needed. The redundant inference steps are eliminated by a backward pass where a set W of needed variables is used for checking the appropriateness of each axiom. The set W contains at the beginning of the backward pass only variables of the goal. When an axiom is classified as useful and kept in the proof, its input variables are included into W and its output variables are excluded from W. This pass is done in linear time with respect to the size of the proof.

3.5.3 Selection of conditional computability statements

A conditional computability statement can be applied, iff all its inputs are given, all its subtasks are solvable and its output is not yet derived. This leads us to the goal-directed search on an and-or tree of subtasks. An and-node represents a computability statement (all its subtasks must be solvable) and an or-node represents a choice between different computability statements. An axiom may be needed in one and the same proof several times (see example proof, Section 3.3). This causes the high complexity of search (provability in IPC is PSPACE complete).

3.5.4 Priority-based selection

This tactic assumes that weights are attached to computability statements. Most usually weights are based on an estimated computational cost of function/program which implements computations specified by the computability statements. This tactic has been implemented only experimentally, because getting the proper weights is difficult in practice.

3.5.5 Splitting the set of axioms

This tactic requires partitioning of the set of axioms. A computability statement is selected only from a specified partition. The partitioning is usually done on the level of the specification language. This tactic is widely used in the NUT system and it is a feature of object-orientation rather than of the logical part of the system. In particular, it is used in independent subtasks discussed below.

4 A SPECIFICATION LEVEL

The $LL1$ language is the internal language for structural synthesis. More compact languages are needed for writing problem specifications. One of these, the NUT specification language [15], is used in the sequel. It is an object-oriented language whose methods and structural relations are automatically translated into formulae of the $LL1$ language. The essential feature of specification languages for SSP is that they enable us to represent concepts (classes in the object-oriented terminology) as theories, and compose the theories in an easy way. This feature will be used later in this paper in production rules for building new theories. The interested reader can find a full description of NUT in [18] as well as from *http://www.it.kth.se/labs/se/nut.html*.

4.1 A specification language

Here we only briefly present some minimal set of the NUT language constructions, which are necessary for understanding presented examples.

The NUT language contains six predefined classes: <u>bool</u>, <u>numeric</u>, <u>text</u>, <u>array</u>, <u>program</u>, <u>any</u>. The last is a universal class, which should be narrowed to any well-defined class before an object of this class is used.

All other classes are specified as collection of declarations of the following form:

Superclass declaration

```
super C;
```

where `C` is a superclass name

Component declaration

```
var A : C;
```

where `A` is a new component of class `C`

Method declaration

```
rel R : <axiom> {<program>};
```

where `R` is a new method (the axiom name `R` may be omitted.). The `<axiom>` consists of lists of input and output parameters of the `<program>` separated by `->`. Input parameters may include other axioms which are enclosed into square brackets - such parameters are called subtasks.

Equivalence declaration

```
rel A = B;
```

where `A` and `B` are components with the same value

Equation declaration

```
rel e1 = e2;
```

where `e1, e2` are arithmetic expressions which define a constraint on the variables occurring in them

We are going to use some syntactic extensions of the above-mentioned specification language, which are presented below. Here D1, .., Dn denote declarations.

```
var D1;...; Dn;    ==      var D1;...; var Dn;
rel D1;...; Dn;    ==      rel D1;...; rel Dn;
var A : C x1 = y1,...,xn = yn;        ==
                   var A : C; rel A.x1 = y1; ...;A.xn = yn;
```

where `y1,..., yn` are objects in the current context.

Statements of the NUT language are translated into $LL1$ language according to transformation rules. Here we present only a sketch of transformations we use in the examples below.

Transformation 4.1. (Object variables) With any object variable x (which is a class instance or a class component) we associate a propositional variable to be denoted by X. For objects with compound names: $x.a\dots.b$ it is $X.A\dots.B$

The following standard functions are defined in the NUT system. They can be used as realizations of the $LL1$ formulae:

$constr(m)$ - forming a compound object from its m components

$select_i(obj)$ - producing components from the given object obj

eq - equality function

Transformation 4.2. (Compound objects) For each object x corresponding to a class component an unconditional computability statement is introduced specifying how to compute it from the object of the class it belongs to. Realization of this statement is $select_i(x)$ where i is a number of the component.

For each compound object an unconditional computability statement is introduced specifying how to compute it from its components. Realization of this statement is $constr(m)$ where m is a list of the object's components.

Transformation 4.3. (Axioms without subtasks) An axiom without subtasks is transformed into a set of unconditional computability statements (one for each output variable) with one and the same set of input variables. Realization of the resulting $LL1$ formulae is expressed by the <program>.

Transformation 4.4. (Axioms with subtasks) Axioms with subtasks (see the method declaration above) are transformed into the set of conditional computability statements (corresponding to each output variable of the program) where single input variables stay on the left-hand side of the implication, output variables on the right-hand side and subtasks are presented as nested implications in the formulae. Realization of the resulting $LL1$ formulae is expressed by the <program> with parameters corresponding to the subtasks.

4.2 The Minimax problem specification

A working example in this paper is the minimax problem.

Let us consider a specification for finding the element in a matrix which has the minimal value among all maximal elements of rows. In order to describe the problem we describe three classes: `matrix`, `min` and `max`. A program for computation of `minimax` is synthesized automatically.

In order to illustrate the whole process of problem solving (from description to program synthesis) we provide each class description with a set of formulae of $LL1$ language extracted from this description.

```
matrix
    var
        elem, row, column: numeric;
        mval:any;
    rel
        get: mval,row,column -> elem {fget()};
```

A matrix contains elements organized into rows and columns. In the class description we do not consider internal representation of the matrix (it can be two-dimensional array, file or may have another representation). We assume that program `fget`, which can be written in NUT or C as a traditional imperative program, takes care about proper manipulation of elements of matrix.

The `matrix` class is transformed into the following set of $LL1$ formulae.

$$MVAL, ROW, COLUMN \xrightarrow[get]{} ELEM$$

$$MVAL, ROW, COLUMN, ELEMENT \xrightarrow[constr]{} MATRIX$$

$$MATRIX \xrightarrow[select1]{} ELEM$$

$$MATRIX \xrightarrow[select2]{} ROW$$

$$MATRIX \xrightarrow[select3]{} COLUMN$$

$$MATRIX \xrightarrow[select4]{} MVAL$$

We can see that methods correspond to computability statements and structural relations correspond to standard constructor and selector statements. In order to simplify the example we do not consider structural relations in other classes.

```
max
   var
      arg, funval, maxval: numeric;
   rel
      max: [arg -> funval] -> maxval{fmax()};
```

The `max` class has a method for computing maximum of an unary function specified by $arg \rightarrow funval$. The corresponding formula is as follows.

$$(ARG \underset{\phi}{\rightarrow} FUNVAL) \xrightarrow[fmax]{} MAXVAL$$

```
min
   var
      arg, funval, maxval: numeric;
   rel
      min: [arg -> funval] -> minval{fmin()};
```

The `min` class has a method for computing minimum of an unary function specified by $arg \rightarrow funval$, and the corresponding formula is

$$(ARG \underset{\phi}{\rightarrow} FUNVAL) \xrightarrow[fmin]{} MINVAL$$

```
minimax
      var
         m: matrix;
         maxinrow: max arg=m.column,funval=m.elem;
         minimax: min arg=m.row,funval=maxinrow.maxval;
```

The `minimax` class specifies our problem for computation of minimal element from maximal elements on matrix's columns. Symbol "=" denotes equality relations which are transformed into formulae of $LL1$ with a standard eq function which implements them. The corresponding set of $LL1$ formulae is as follows.

$$M.MVAL, M.ROW, M.COLUMN \xrightarrow[get]{} M.ELEM$$

$$(MAXINROW.ARG \underset{\phi}{\rightarrow} MAXINROW.FUNVAL) \xrightarrow[fmax]{} MAXINROW.MAXVAL$$

$$(MINIMAX.ARG \underset{\phi}{\rightarrow} MINIMAX.FUNVAL) \xrightarrow[fmin]{} MINIMAX.MINVAL$$

$$MAXINROW.ARG \xrightarrow[eq]{} M.COLUMN$$

$$M.COLUMN \underset{eq}{\longrightarrow} MAXINROW.ARG$$

$$MAXINROW.FUNVAL \underset{eq}{\longrightarrow} M.ELEM$$

$$M.ELEM \underset{eq}{\longrightarrow} MAXINROW.FUNVAL$$

$$MINIMAX.FUNVAL \underset{eq}{\longrightarrow} MAXINROW.MAXVAL$$

$$MAXINROW.MAXVAL \underset{eq}{\longrightarrow} MINIMAX.FUNVAL$$

$$MINIMAX.ARG \underset{eq}{\longrightarrow} M.ROW$$

$$M.ROW \underset{eq}{\longrightarrow} MINIMAX.ARG$$

A problem to be solved is described as a method with a keyword `spec` in the implementation part

```
problem: m.val -> minimax.minval {spec}
```

This method is translated into the following theorem to be proven

$$\vdash M.MVAL \to MINIMAX.MINVAL$$

And the proof is

$$\cfrac{\cfrac{\cfrac{\begin{array}{c} M.MVAL(mval) \vdash M.MVAL(mval); M.COLUMN(column) \vdash M.COLUMN(column); \\ \vdash M.MVAL \& M.ROW \& M.COLUMN \underset{get}{\longrightarrow} M.ELEM; M.ROW(row) \vdash M.ROW(row); \end{array}}{\begin{array}{c} \vdash (MAXINROW.ARG \underset{\phi}{\rightarrow} MAXINROW.FUNVAL) \underset{fmax}{\longrightarrow} MAXINROW.MAXVAL; \\ M.ELEM \underset{eq}{\longrightarrow} MAXINROW.FUNVAL; MAXINROW.ARG \underset{eq}{\longrightarrow} M.COLUMN; \\ M.MVAL(mval), M.ROW(row), M.COLUMN(column) \vdash M.ELEM(get(mval, row, column)); \end{array}}(\to -)}{\begin{array}{c} M.MVAL(mval), M.ROW(row) \vdash MAXINROW.MAXVAL(fmax(\lambda column.(get(mval, row, column))); \\ MINIMAX.ARG \underset{eq}{\longrightarrow} M.ROW; MAXINROW.MAXVAL \underset{eq}{\longrightarrow} MINIMAX.FUNVAL; \\ (MINIMAX.ARG \underset{\phi}{\rightarrow} MINIMAX.FUNVAL) \underset{fmin}{\longrightarrow} MINIMAX.MINVAL; \end{array}}(\to --)}{}(\to --)$$

$$\frac{M.MVAL(mval) \vdash MINIMAX.MINVAL(fmin(\lambda row.fmax(\lambda column.get(mval, row, column))));}{\vdash M.MVAL \xrightarrow[\lambda mval.fmin(\lambda row.fmax(\lambda column.get(mval,row,column)))]{} MINIMAX.MINVAL;}(\rightarrow +)$$

Q.E.D.

The extracted program is:

$$\lambda mval.fmin(\lambda row.fmax(\lambda column.get(mval, row, column)))$$

This example shows as well how unsuitable the logic of SSP is for manual synthesis of programs. Luckuly, this logic can be used completely automatically. In this case we are able without difficulties to handle thousands of axioms.

5 OBJECT-ORIENTED COMPUTING AND INDEPENDENT SUBTASKS

Several strategies are related to object-oriented computing. Here we consider the following two strategies which are widely used for problem solving:

run-time synthesis

independent subtasks

We have to point out that only dependent subtasks fit into the IPC. Other kinds of subtasks considered in this section are related to different calculi, although technically the changes in implementation are small. While synthesis is not IPC-based here it shares the same specification language and is often used as a complement to the basic synthesis.

5.1 Run-time synthesis

The idea of run-time synthesis of programs can be described as follows. A problem to be solved is expressed as a method with keyword `spec` in the implementation part. This method can be explicitly invoked, or it can be used in a synthesized program as any other method of a class. Calling such a method invokes the program synthesizer which works as it is described in Sect. 2. If the synthesis is successful, then a program is generated and computations are performed. It may happen that a method in the synthesized program is another method with keyword `spec` in the implementation part, or it calls a method with keyword `spec` again. A final executed program will contain a hierarchy of synthesized programs calls.

The advantages of the run-time synthesis are dynamic handling of problems to be solved by the synthesizer, and explicit calls of synthesizer in the manually written program texts. The latter allows to combine declarative and imperative styles of programming. A useful practical feature of the run-time synthesis is usage of values of objects as default axioms. For example, if an object `matrix` gets a value *matrix*, then the corresponding axiom $\vdash MATRIX(matrix)$ is added. This allows us to take into account all recent computations as default inputs for program synthesis.

The run-time synthesis also enables one to reduce the complexity of synthesis by eliminating the usage of the inference rule $(\rightarrow --)$. We call the synthesis with rules $(\rightarrow -)$ and $(\rightarrow +)$ only *linear synthesis* because it has linear time complexity. In this case, nested implications are replaced by explicit calls to synthesizer in bodies of programs. This strategy works well in many practical cases. However it doesn't allow to model SSP because of its logical incompleteness. This can be illustrated by the following example. Let us assume that we have the following methods:

```
method1: in1, in2 -> out {out:=method4(in1,in2)};
method2: in1, in2 -> out {out:=method3(in1,in2)};
method3: a, b -> c {spec};
method4: x, y -> z {spec};
method5: a, b -> c {c:=a+b};
problem: in1, in2 -> out {spec};
```

An invocation of the method `problem` may result in synthesis of a program which contains invocation of `method1`. However execution of `method1` will fail because of unsolvability of the problem described by `method4`. At the same time invocation of `method2` could be performed successfully.

By using nested implications the problem above could be described in the following solvable way:

```
method1: [x, y -> z], in1, in2 -> out {spec};
method2: [a, b -> c], in1, in2 -> out {spec};
method5: a, b -> c {c:=a+b};
problem: in1, in2 -> out {spec};
```

We would like to notice that in spite of incompleteness of the run-time synthesis it is widely used for specification of problems. However, it requires a better understanding of a process of synthesis from a user.

5.2 Independent subtasks

The strategies described here reduce a complexity of the general SSP method by introducing independent contexts for synthesis of subtasks.

Let us have a formula of the $LL1$ language

$$(A \to B) \to (C \to D)$$

In the proof of the subtask $(A \to B)$, i.e. in the derivation of $A \vdash B$ one can use all results (all variables derived) at the time the subtask is used. In the case of independent subtask the results of synthesis obtained before starting a subtask are not available. In other words, each independent subtask is solved independently from others and from top level synthesis process. This strategy is implemented in the NUT system for the subtasks whose context is explicitly shown by a class name `c` as follows `[c:- A -> B]`. It means that in this case a set of axioms for deriving $A \to B$ is defined only by the specification of the class `c`.

In the earlier SSP systems a weaker notion of independency is implemented. For example, in the PRIZ system subtasks are independent from each other but not from the top level synthesis process. It means that only premises of other subtasks can not be used in proof of the current subtask. It was proved that planning with such independent subtasks leads to sound and correct derivation algorithm for the modal logic S4 (see [11]). In addition to reducing complexity, independent subtasks can be seen as a natural way for expressing problems in a modular way.

Let us consider a simulation problem. Many systems can be represented by a model of simulated object and a process of simulation. Simulation object can be described by a set of building blocks and connections between the blocks. Processes which occur in blocks are specified inside the blocks. We must specify a concept of the process for the system as a whole.

The following class is a simple specification of a process applicable in simulation.

```
process
       var
          result: any;
          from, to, timeStep : numeric;
          initstate, state, nextstate: any;
       rel
          procrel: [model:- state -> nextstate],
                   from, to, timeStep, initstate -> result {...};
```

The method `procrel` uses an independent subtask to be solved on the `model` class. The `model` class describes a particular object whose behavior has to be simulated.

6 PARTIAL DEDUCTION AS A STRATEGY FOR SSP

The main motivation for development of partial deduction framework of SSP is to avoid the difficulties with expressing some goals in $LL1$.

Our experience with the SSP systems shows that users are not always able to completely specify a goal in the form of $\vdash \underline{A} \to \underline{G}$ but rather prefer to indicate only known variables ($\underline{A}$) of the problem. Then the users are quite satisfied with result that everything that is computable from the known (and initialized) variables is computed. In other words it corresponds to the goal: "compute everything that is possible to compute". These goals are not efficiently expressible in $LL1$ because of the necessity to take into account the set of all variables of the specification and all subsets of this set. Additionally, an order over the subsets would have to be defined and possible output parameters considered as a disjunction of the subsets. This is a very inefficient approach.

On the other hand, partial deduction allows us to make efficient derivations in $LL1$, even when a goal is not specified completely.

The paper [9] applies the principle of partial deduction (see [6, 3] for more details) to SSP and proves its correctness and completeness in the SSP framework.

Here we briefly give only an idea of partial deduction for SSP in the case of unconditional computability statements by describing some essential definitions and theorems. Complete set of definitions, theorems and proofs as well as more complicated case of conditional computability statements are considered in [9].

First we introduce a notion of unconditional resultant.

Definition 6.1. **(**$LL1$ **unconditional resultant)** An $LL1$ **unconditional resultant** is an unconditional computability statement $\underline{A} \xrightarrow[\lambda \underline{a}.f]{} G$ where f is a term representing the function which computes G from potentially composite functions over $a_1, \ldots, a_n$

Then we can define derivation of unconditional resultant as follows. In our definitions we use the notion of selection function (or computation rule from logic programming [5]). We denote a substitution of a by b in a term f as $f[b/a]$.)

Definition 6.2. **(Derivation of an** $LL1$ **unconditional resultant)** Let $\Re$ be a fixed selection function and P a set of LL1 axioms. A **derivation of an** $LL1$ **unconditional resultant** R_0 is a finite sequence of $LL1$ resultants: $R_0 \Rightarrow_{\Re} R_1 \Rightarrow_{\Re} R_2 \Rightarrow_{\Re} \ldots$, where,

(1) for each j, R_j is an unconditional computability statement of the form

$$\underline{B_1} \& \ldots \& B_i \& \ldots \& \underline{B_n} \xrightarrow[\lambda \underline{b_1} \ldots b_i \ldots \underline{b_n}.f]{} G$$

and R_{j+1} (if any) is of the form:

$$\underline{B_1}\&...\&\underline{C}\&...\&\underline{B_n} \quad \xrightarrow[\lambda \underline{b_1}...\underline{c}...\underline{b_n}.f[h(\underline{c})/b_i]]{} \quad G$$

if

the $\Re$-selected atom in R_j is B_i called the matching atom and there exists an unconditional computability statement $\underline{C} \underset{h}{\rightarrow} B_i \in P$ called the matching computability statement.

(2) for each j, R_j is an unconditional computability statement of the form

$$\underline{A_1}\&...\&\underline{A_i}\&...\&\underline{A_m} \xrightarrow[\lambda \underline{a_1}...\underline{a_i}...\underline{a_m}.f]{} V$$

and R_{j+1} (if any) is of the form:

$$\underline{A_1}\&...\&\underline{A_i}\&...\&\underline{A_m} \quad \xrightarrow[\lambda \underline{a_1}...\underline{a_i}...\underline{a_m}.h[f(\underline{a_1},...,\underline{a_i},...,\underline{a_m})/v]]{} \quad H$$

if

the $\Re$-selected atoms in R_j are $\underline{A_i}$ and/or $\underline{V}$ called the matching atoms. $\underline{\text{V}}$ denotes a conjunction of V (heads of computability statements) from resultants $R_0, ..., R_j$
there exists an unconditional computability statement $\underline{A_i}\&\underline{V} \quad \underset{h}{\rightarrow} \quad H \in P$ called the matching computability statement.

(3) If R_{j+1} does not exist, R_j is called a leaf resultant.

Definition 6.2 gives two ways of deriving resultants. (1) corresponds to backward derivation and (2) corresponds to forward derivations.

Definition 6.3. (Partial deduction of a propositional goal) A **partial deduction of a propositional goal** $\underline{A} \rightarrow G$ in a specification P is the set of leaf resultants derived from $G \rightarrow G$ (derivation form (1) and in this case $\underline{A}$ cannot be selected atoms) or from $\underline{\underline{A} \rightarrow A}$ (derivation form (2) and in this case G cannot be selected atom).

The notions of correctness and completeness are defined as follows. Let P be an $LL1$ specification, $\underline{A} \rightarrow G$ a propositional goal, and P' a partial deduction of P wrt to $\underline{A} \rightarrow G$.

Definition 6.4. (Correctness of partial deduction of an $LL1$ specification) The function (or its computational equivalent) for the computation of $\underline{A} \rightarrow G$ is derivable (can be extracted from a proof of $\underline{A} \rightarrow G$) from P if it is derivable from P'.

Completeness is the converse:

Definition 6.5. (Completeness of partial deduction of an $LL1$ specification) The function (or its computational equivalent) for the computation of $\underline{A} \to G$ is derivable (can be extracted from a proof of $\underline{A} \to G$) from P' if it is derivable from P.

The following Lemmata and theorem has been proven.

Lemma 6.1. Derivation form (1) of $LL1$ unconditional resultants is a derivation by SSP inference rules.

Lemma 6.2. Derivation form (2) of $LL1$ unconditional resultants is a derivation by SSP inference rules.

Theorem 6.1. (Correctness and Completeness of PD in $LL1$ unconditional specification) Partial Deduction of a $LL1$ unconditional specification is correct and complete.

Corresponding Lemmata and theorems have been proven also for conditional resultants and different derivation forms (see [9]).

In order to summarize our results we can say that partial deduction can be used as s correct and complete strategy for automatic deduction in SSP.

In order to demonstrate a partial deduction for unconditional computability statements we consider the following very simple example.

Let us have the following computability statements as a problem specification:

$$A \underset{f}{\to} B,\ B \underset{h}{\to} C,\ C \underset{g}{\to} D$$

then some possible sequence of resultants with forward derivation can be as follows:

1. $A \xrightarrow[\lambda a.h(f(a))]{} C$
2. $A \xrightarrow[\lambda a.g(h(f(a)))]{} D$
3. $B \xrightarrow[\lambda b.g(h(b))]{} D$

and with backward derivation it can be as follows:

1. $B \xrightarrow[\lambda b.g(h(b))]{} D$

2. $A \xrightarrow[\lambda a.g(h(f(a)))]{} D$

3. $A \xrightarrow[\lambda a.h(f(a))]{} C$

By applying partial deduction to computability statements we obtain new computability statements which constitute a specialization of the initial specification. In other words, partial deduction can be considered as performing transformations on a problem specification (in our case on a set of computability statements) with correct and complete transformation rules. We also need to define stopping criteria for partial deduction process.

Generally the following stopping criteria are suggested:

1. *Stepwise*: The users are asked which of the possible transformations they want to perform before each partial deduction step.
2. *Bounded*: The maximum number of steps of the partial deduction process is specified.
3. *Goal-based:* A partial deduction process stops when the resultant computability statement is the goal.
4. *Exhaustive:* A partial deduction process stops when no new transformations can be applied.

The last stopping criteria is the most important for partial deduction, because it makes possible deriving programs for incompletely specified goals.

7 CONCLUDING REMARKS

¿From the presentation above can be seen that SSP is most useful in combination with some high-level specification language. This is how we have used it in the NUT system. Axioms of $LL1$ are explicitly written only for preprogrammed moduli there. Quite a number of axioms can be derived from specifications of data structures as shown in the minimax example. As soon as a class is used as a type specifier, all axioms of the class are automatically reused for the object whose type is being specified. All this leads to logical specifications with large number of axioms. We have solved simulation problems whose logical descriptions contain thousands of axioms, although a high-level specification of such a problem is written manually on a small number of lines. Luckily enough, the SSP inference procedure works almost in linear time for specifications with small number of weakly related subtasks. Hence we are able to handle large number of axioms in our proofs [12].

SSP supports equational programming as well. As soon as there is an equation solver for a class of equations, we can represent any equation $E(x_1, ...x_n) = 0$ of this class logically by writing an axiom $x_1, ..., x_{i-1}, x_{i+1}, ..., x_n \rightarrow x_i$ for each variable x_i which is computable from the equation. Realization for the axiom will be a solving function of the equation for the variable x_i on the right side of the axiom. In this case one has to find minimal simultaneously solvable blocks of equations and represent each block by means of a set of axioms telling which computations are possible on this block. The main difficulty in handling equations by means of SSP is not in logic but in equation solving itself.

There are more extensions of SSP which have been under consideration. In particular, instead of encoding disjunctions as described in [11], one can introduce the disjunction symbol into the language directly, and add introduction and elimination rules for disjunction to the inference rules of SSP. Experiments in this direction are being done by Sven Laemmermann at KTH.

The strategies of structural synthesis of programs have to be made concrete in the form of algorithms. There is a paper [19] where SSP is explained in terms of higher order constraint solving. It contains also hints on data structures and algorithms implementing the general strategy of SSP.

Practical application of SSP for several real problems can be found in [16].

8 ACKNOWLEDGEMENTS

This work was partially financed by Andersen Consulting Research Foundation (project "Agent-Based Methods").

REFERENCES

[1] Dikovsky, A.: Solving algorithmic problems of synthesis of programs without loops in linear time. System Programming and Computer Software, 1985, 3:38–49, 1985.

[2] Kleene, S.: Introduction to metamathematics. Amsterdam, North-Holland, 1952.

[3] Komorowski, J.: A Prolegomenon to partial deduction, *Fundamenta Informaticae*, 18(1):41–64, 1993.

[4] Lämmermann, S.: Automated composition of Java software (Lic. Thesis), TRITA-IT AVH 00:03, ISSN 1403-5286, Teleinformatics/KTH, Stockholm, 2000.

[5] Lloyd, J. W.: Foundations of logic programming. Springer Verlag, second edition, 1987.

[6] Lloyd, J. W. and Shepherdson, J. C.: Partial evaluation in logic programming, *Journal of Logic Programming*, 11:217–242, 1991. also: Technical Report CS-87-09 (revised 1989), University of Bristol, England, July 1989.

[7] Lowry, M. R. and McCartney, R. D. editors.: *Automating Software Design*, California, USA, 1991. The MIT Press.

[8] Manna, Z. and Waldinger, R.: A Deductive approach to program synthesis. ACM Trans. on Programming Languages and Systems, 2(1):294:327, Jan, 1980.

[9] Matskin, M. and Komorowski, J.: Partial Structural Synthesis of Programs, Fundamenta Informaticae, IOS Press, 31(1997) pp. 125-144.

[10] Mints, G. and Tyugu, E.: Justification of structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[11] Mints, G.: Propositional Logic Programming. In J. Hayes, D. Michie, E. Tyugu (eds.) *Machine Intelligence*, v. 12, Clarendon Press, 1991, p. 17 - 37

[12] PAHAPILL, J.: Programmpaket zur modeliering der hydromachinen systeme. 6. Fachtagung Hydraulik und Pneumatik, Magdeburg, pp. 609-617, 1985.

[13] TYUGU, E.: *Knowledge-Based Programming.* Addisson-Wesley and Turing Institute press, 1988.

[14] TYUGU, E.: Three new-generation software environments. *Communications of the ACM*, 34(6):46–59, June 1991.

[15] TYUGU, E.: Classes as program specifications in NUT. *Journal of Automated Software Engineering*, 1, 315 –334, 1996.

[16] TYUGU, E., MATSKIN, M. AND PENJAM, J.: Applications of Structural Synthesis of Programs. In: J. Wing, J. Woodcock, J. Davies (Eds.) FM99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, Sept. 1999. Proc. vol. I, Lecture Notes in Computer Science (LNCS) No. 1708, Springer, 1999, p. 551 - 569.

[17] VOLOZH, B., MATSKIN, M., MINTS, G. AND TYUGU, E.: Theorem proving with the aid of program synthesizer. Cybernetics, 6:63-70, 1982.

[18] UUSTALU, T., KOPRA, U., KOTKAS, V., MATSKIN, M. AND TYUGU, E.: The NUT Language Report. The Royal Institute of Technology (KTH),TRITA-IT R 94:14, 51 p., 1994.

[19] UUSTALU, T. AND TYUGU, E.: Higher-Order Functional Constraint Networks. In *Constraint Programming. NATO ASI Series F*, v. 131, Springer, 1994, pp. 116 - 139.